AviSpectra

Bird Species Identification

Project ⤖ Bird Recognition System with Real-time Images

Abstract

Automated bird species identification is vital in biodiversity conservation, ecological research, and environmental monitoring. This work presents a deep learning approach for classifying Indian bird species using the **Indian-Birds-Species-Image-Classification** dataset, which includes 37,500 labeled images of 25 bird species in India. We develop an **effective Custom CNN Model** for extracting complex patterns from images. This study demonstrates deep learning's effectiveness in accurately classifying bird species and suggests applications for automated wildlife monitoring, conservation, and education.

Introduction

Bird species classification is important in biodiversity conservation, ecological research, and wildlife monitoring. Traditional identification methods are labor-intensive and require domain expertise. With advancements in computer vision and deep learning, it is now possible to automate bird identification tasks effectively. This project utilizes deep learning methods to classify images of Indian bird species, contributing to faster, more accurate species recognition in various fields.

Keywords → Bird Species Classification, Deep Learning, Image Classification, Computer Vision

Practical Usage

The developed CNN model can be applied in:

1. Supporting researchers in quick and accurate bird species identification.
2. Assisting conservation efforts by tracking and monitoring bird populations.
3. Powering mobile and web applications for birdwatchers and environmental studies.
4. Enhancing automated wildlife monitoring and data collection in natural habitats.

Data Availability

The *dataset* (https://www.kaggle.com/datasets/ichhadhari/indian-birds) contains images(augmentated - Horizontal flip-flop, Rotation ± 30, Brightest ±3%) of the following 25 bird species:

Asian Green Bee-eater	Brown-Headed Barbet	Cattle Egret	Common Kingfisher
Common Myna	Common Rosefinch	Common Tailorbird	Coppersmith Barbet
Forest Wagtail	Gray Wagtail	Hoopoe	House Crow

Indian Grey Hornbill
Jungle Babbler
Rufous Treepie
Indian Roller
Indian Peacock
Northern Lapwing
Sarus Crane
Indian Pitta
Ruddy Shelduck
White Wagtail
White-Breasted Waterhen
Red-Wattled Lapwing
White-Breasted Kingfisher

The dataset includes **37,500 images** in total, split into:

TRAINING SET	VALIDATION SET	TESTING SET
24000	6000	7500

Each species has 1,500 images, making the dataset well-balanced for classification tasks. The dataset is highly suitable for training and evaluating machine learning models focused on image-based species recognition.

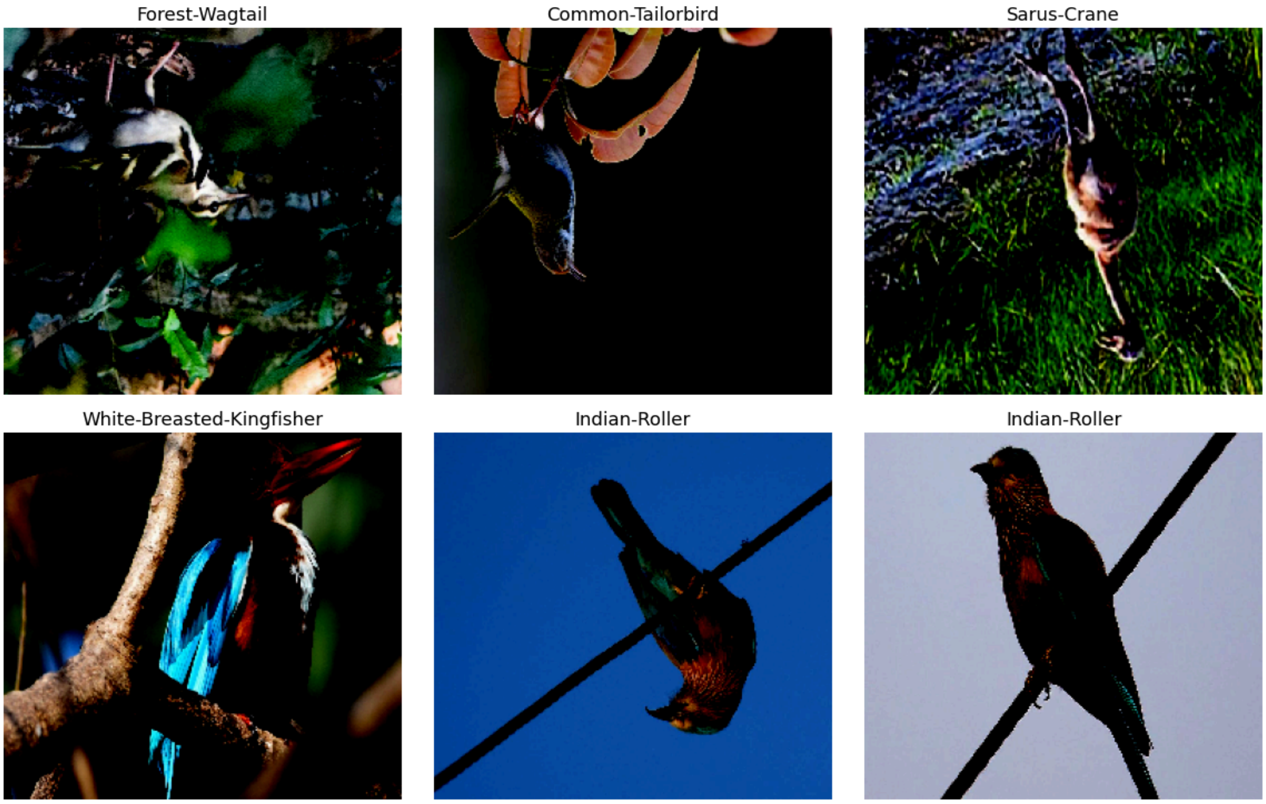


Figure 1: Sample Dataset Images

Hardware/Software Requirements

Hardware:

1. Minimum 8 GB RAM (>8 GB preferred)
2. At least 6 GB of free storage space

Software:

1. Python 3.10 or higher
2. Deep learning frameworks: TensorFlow or PyTorch
3. Supporting libraries: NumPy, Pandas, Matplotlib, Scikit-learn
4. Jupyter Notebook or any preferred Python IDE

Methodology

Model Architecture

The proposed model (Custom CNN) architecture leverages the DenseNet121 backbone pre-trained on ImageNet, with the top classification layers removed to enable transfer learning. All layers of the base model are frozen to preserve learned features and reduce computational complexity. A lightweight classification head is appended, consisting of a Global Average Pooling layer to flatten feature maps, a dense layer with 128 ReLU-activated units, and a dropout layer (rate = 0.2) for regularization. The final output layer employs a softmax activation function to predict class probabilities across 25 target categories. The model is optimized using the Adam optimizer with a categorical cross-entropy loss function and evaluated using accuracy, precision, and recall metrics.

Project Structure

```
── app.py                # Main Streamlit application handling user interface and logic
── model/                # Pre-trained deep learning models for bird species detection
── ai_agent.py           # AI agent to retrieve detailed bird information using Google Gemini
── utils.py              # Utility functions for processing and UI management
── requirements.txt      # List of required Python libraries
```

System Workflow

The bird detection system follows a structured workflow as given in Figure 2, involving user interaction, data processing, model detection, and information retrieval:

1. User Interface:
   Users can upload an image, capture a photo through their webcam, or upload/stream a video using the website. These options allow flexible inputs to the system for bird detection.

2. Data Processing:
   Uploaded images are pre-processed into 256x256 pixels, greyscaled to avoid colouration errors, and passed for feature extraction. Videos are divided into frames at 20 frames per second, ensuring detailed inspection for bird detection across time.

3. Bird Detection Models:
   Processed inputs are passed to specific deep learning models. Separate models are used for image, camera, and video frame detection to ensure optimized predictions.

4. Results Display and AI Interaction:
   The system displays the predicted bird species along with a confidence score. Additionally, AI agents are integrated for each input type to provide users with detailed bird information upon request.

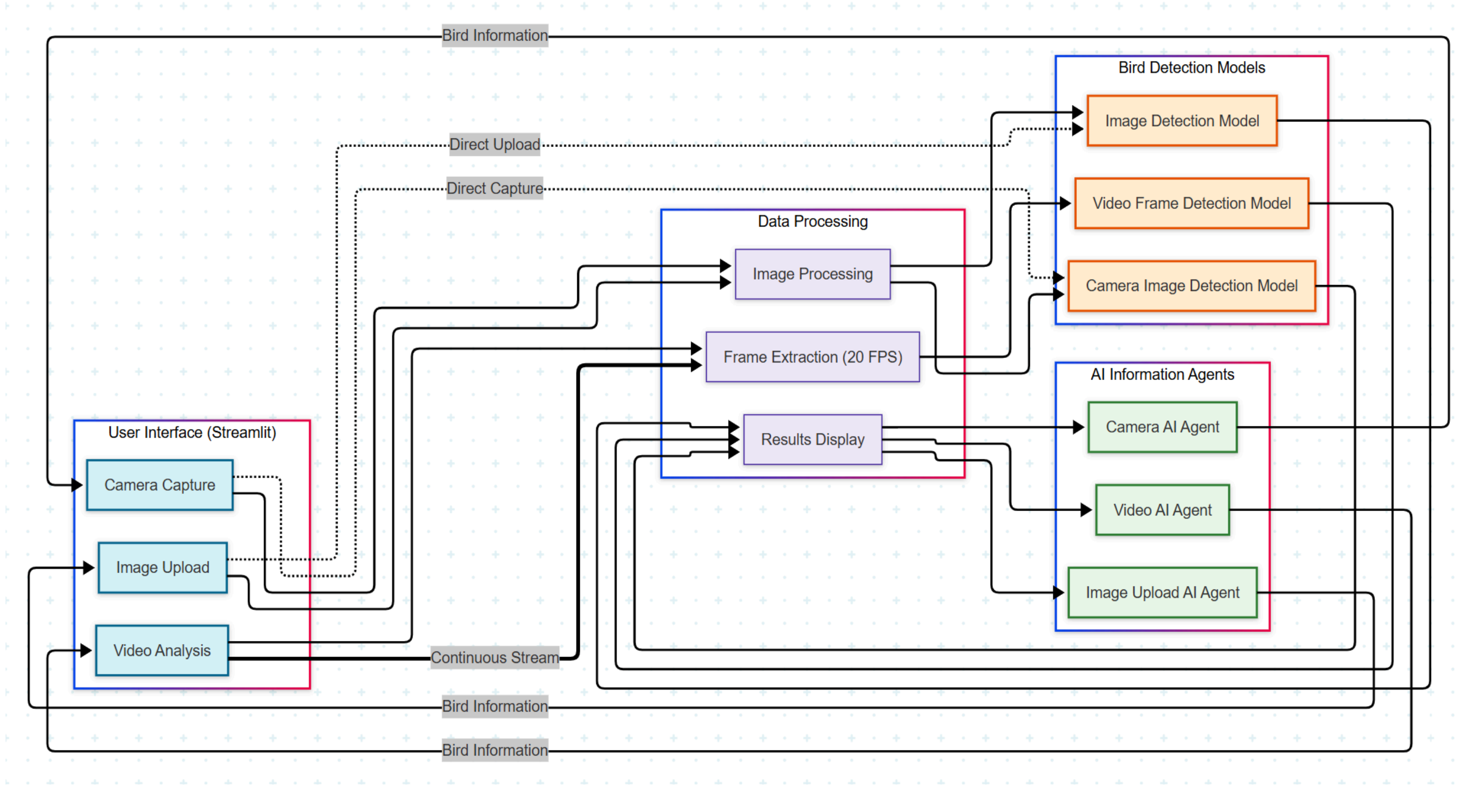


Figure 2: Website Architectural Workflow

Comparative Study

Training Evaluation of Classification Models:

To assess the efficiency of various deep learning architectures, we evaluated five models—DenseNet121, ResNet50V2, MobileNetV2, Custom CNN-1, and Custom CNN-2—based on training, validation, and testing accuracy, loss values, and total training time.

MODEL	ACCURACY			LOSSES		TIME TAKEN (in sec)
	Training	Validation	Testing	Training	Validation	
DenseNET 121	0.9999	0.9428	0.9663	0.0074	0.2223	5370.82
ResNET 50 V2	0.9972	0.8917	0.9195	0.0302	0.4108	5284.52
Mobile NET V2	0.9996	0.9235	0.9445	0.0152	0.2941	4544.36
Custom CNN-1	0.9766	0.8205	0.8719	0.0866	0.6858	5454.04
Custom CNN-2	0.9886	0.9437	0.9661	0.0438	0.1851	5095.24

Table 1: Building Metrics and Time of Completion Comparison

Inference

- DenseNet121 achieved the highest testing accuracy (96.63%) with minimal training (0.0074) and validation loss (0.2223), demonstrating superior generalization.

- Custom CNN-2 showed comparable performance (96.61% testing accuracy) with slightly higher training time.
- MobileNetV2 balanced performance and speed, achieving 94.45% testing accuracy with the lowest training time (4544.36 sec).

Performance Metrics:

To evaluate the classification performance of the proposed architecture, overall accuracy, recall, precision, and F1 scores were selected as the accuracy performance metrics. The overall accuracy is the ratio of the total number of correctly classified samples to the total number of samples of all classes. In this study, the samples are blocks extracted from hyperspectral images. Recall, precision, and F1 scores can be calculated from the true positives (TP), the true negatives (TN), the false positives (FP), and the false negatives (FN). The metrics were calculated as follows:

$$\text{Recall} = \frac{TP}{TP+FN} \qquad \text{Precision} = \frac{TP}{TP+FP} \qquad \text{F1 Score} = \frac{2 \cdot TP}{2 \cdot TP+FP+FN}$$

MODEL	TRAINING			VALIDATION			TESTING		
	Precision	Recall	F1-Score	Precision	Recall	F1-Score	Precision	Recall	F1-Score
DenseNET 121	0.9999	0.9996	0.9998	0.9500	0.9378	0.9439	0.9728	0.9632	0.9680
ResNET 50 V2	0.9990	0.9950	0.9980	0.9100	0.8813	0.8954	0.9364	0.9112	0.9236
Mobile NET V2	0.9999	0.9990	0.9994	0.9353	0.9187	0.9240	0.9525	0.9392	0.9458
Custom CNN-1	0.9842	0.9676	0.9763	0.8516	0.8017	0.8258	0.8959	0.8568	0.8759
Custom CNN-2	0.9918	0.9846	0.9882	0.9568	0.9387	0.9472	0.9754	0.9605	0.9681

Table 2: Performance Metrics Comparison on the 7500 Testing Images

Inference

- DenseNet121 achieved near-perfect training performance (F1-score: 0.9998) and demonstrated strong generalization with validation and testing F1-scores of 0.9439 and 0.9680, respectively.

- MobileNetV2 showed a strong balance between accuracy and efficiency, with a perfect training precision (0.9999) and a competitive testing F1 Score of 0.9458.

- Custom CNN-2 delivered comparable performance to DenseNet121, attaining a testing F1-score of 0.9681 and a validation F1-score of 0.9472, reflecting its robustness.

These results indicate that while DenseNet121 leads in consistent performance, MobileNetV2 is efficient, and **Custom CNN-2** stands out as a *reliable, lightweight* alternative for accurate classification.

Source Code/Output

1. Configuration

```
IMG_IND = 256
IMG_SHAPE = (IMG_IND,IMG_IND,3)
IMG_SIZE = (256,256)
BATCH_SIZE = 100
SEED = 10
EPOCHS = 20
PATIENCE = 4
```

2. Callback Function for Model Training

```
def get_callback(model_name):
    callbacks = []

    checkpoint = keras.callbacks.ModelCheckpoint(
                    filepath=f'model.{model_name}.keras',
                    verbose = 1, monitor='val_loss',mode='min')
    callbacks.append(checkpoint)
    anne = keras.callbacks.ReduceLROnPlateau(monitor='val_loss', patience=PATIENCE,
                                             verbose=1, min_lr=0.0000001)
    callbacks.append(anne)
    earlystop = keras.callbacks.EarlyStopping(monitor='loss', patience=2*PATIENCE)
    callbacks.append(earlystop)

    return callbacks
```

3. Transfer Learning Model Using DenseNet121 with Custom Classification Head

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers

# Base DenseNet-121 model (without top classification layer)
base_model = keras.applications.DenseNet121(
    include_top=False,    # Exclude the final dense layer
    weights='imagenet',  # pre-trained weights
    input_shape=IMG_SHAPE
)

# Freeze the base model layers --> retain pre-trained weights
base_model.trainable = False
# Architecture
inputs = keras.Input(shape=IMG_SHAPE)
# Input through the DenseNet121 backbone
x = base_model(inputs)
# Adding lightweight layers at the end for efficiency
x = layers.GlobalAveragePooling2D()(x)   # Convert the 2D features to 1D feature vector
# Adding a lightweight fusion pathway (fully connected layer)
x = layers.Dense(128, activation='relu')(x)
x = layers.Dropout(0.2)(x)   # Dropout for regularization
# Output layer
outputs = layers.Dense(25, activation='softmax')(x)
# Final model
model5 = keras.Model(inputs=inputs, outputs=outputs)

# Compiling
```

```
model5.compile(
    optimizer=keras.optimizers.Adam(learning_rate=0.001),
    loss='categorical_crossentropy',
    metrics=[
        'accuracy',
        keras.metrics.Precision(name='precision'),
        keras.metrics.Recall(name='recall')
    ]
)
model5.summary()
```

Model: "functional_2"

Layer (type)	Output Shape	Param #
input_layer_3 (InputLayer)	(None, 256, 256, 3)	0
densenet121 (Functional)	(None, 8, 8, 1024)	7,037,504
global_average_pooling2d_4 (GlobalAveragePooling2D)	(None, 1024)	0
dense_6 (Dense)	(None, 128)	131,200
dropout_2 (Dropout)	(None, 128)	0
dense_7 (Dense)	(None, 25)	3,225

Total params: 7,171,929 (27.36 MB)
Trainable params: 134,425 (525.10 KB)
Non-trainable params: 7,037,504 (26.85 MB)

4. Model Training with Callbacks and Time Tracking

```
# Define callbacks
callbacks = get_callback('customcnn')
# Train the model
start_time = time.time()
history2 = model5.fit(
    train_df,
    epochs= EPOCHS,
    validation_data=valid_df,
    callbacks=[callbacks]
)
end_time = time.time()
```

```
Epoch 1/20
240/240 ━━━━━━━━━━━━━━━━━━━━ 0s 785ms/step - accuracy: 0.9114 - loss: 0.3152 -
precision: 0.9537 - recall: 0.8733
Epoch 1: saving model to model.customcnn.keras
240/240 ━━━━━━━━━━━━━━━━━━━━ 273s 1s/step - accuracy: 0.9114 - loss: 0.3152 -
precision: 0.9537 - recall: 0.8733 - val_accuracy: 0.9333 - val_loss: 0.2337 - val_precision:
0.9650 - val_recall: 0.9057 - learning_rate: 0.0010
.
.
.
Epoch 20/20
240/240 ━━━━━━━━━━━━━━━━━━━━ 0s 784ms/step - accuracy: 0.9886 - loss: 0.0438 -
precision: 0.9918 - recall: 0.9846
Epoch 20: saving model to model.customcnn.keras
240/240 ━━━━━━━━━━━━━━━━━━━━ 246s 1000ms/step - accuracy: 0.9886 - loss: 0.0438 -
precision: 0.9918 - recall: 0.9846 - val_accuracy: 0.9437 - val_loss: 0.1851 - val_precision:
0.9568 - val_recall: 0.9387 - learning_rate: 1.0000e-06
```

5. Model Evaluation and Prediction on Test Data

```
def prediction(test_imgs, model):
    list_of_prediction = []
    labels = []
    for i in range(len(test_imgs)):
        imgs, label = test_imgs[i]
        pred = model.predict(imgs)
        for i in range(len(pred)):
            list_of_prediction.append(pred[i].argmax())
            labels.append(label[i].argmax())

    return list_of_prediction, labels

training_time = end_time - start_time
test_loss, test_accuracy, test_precision, test_recall = model5.evaluate(test_df)
predictions , test_labels = prediction(test_df, model5)
predictions = np.array(predictions)
test_labels = np.array(test_labels)
```

6. Evaluation Metrics and Save the Trained Model

```
print(f"Total Training Time: {training_time:.2f} seconds\n")
print(f"Test Accuracy: {test_accuracy:.4f}\n")
print(f"Test Precision: {test_precision:.4f}\n")
print(f"Test Recall: {test_recall:.4f}\n")
print(classification_report(test_labels,predictions))

model5.save('customcnn_trained_model.h5') # Save the model
```

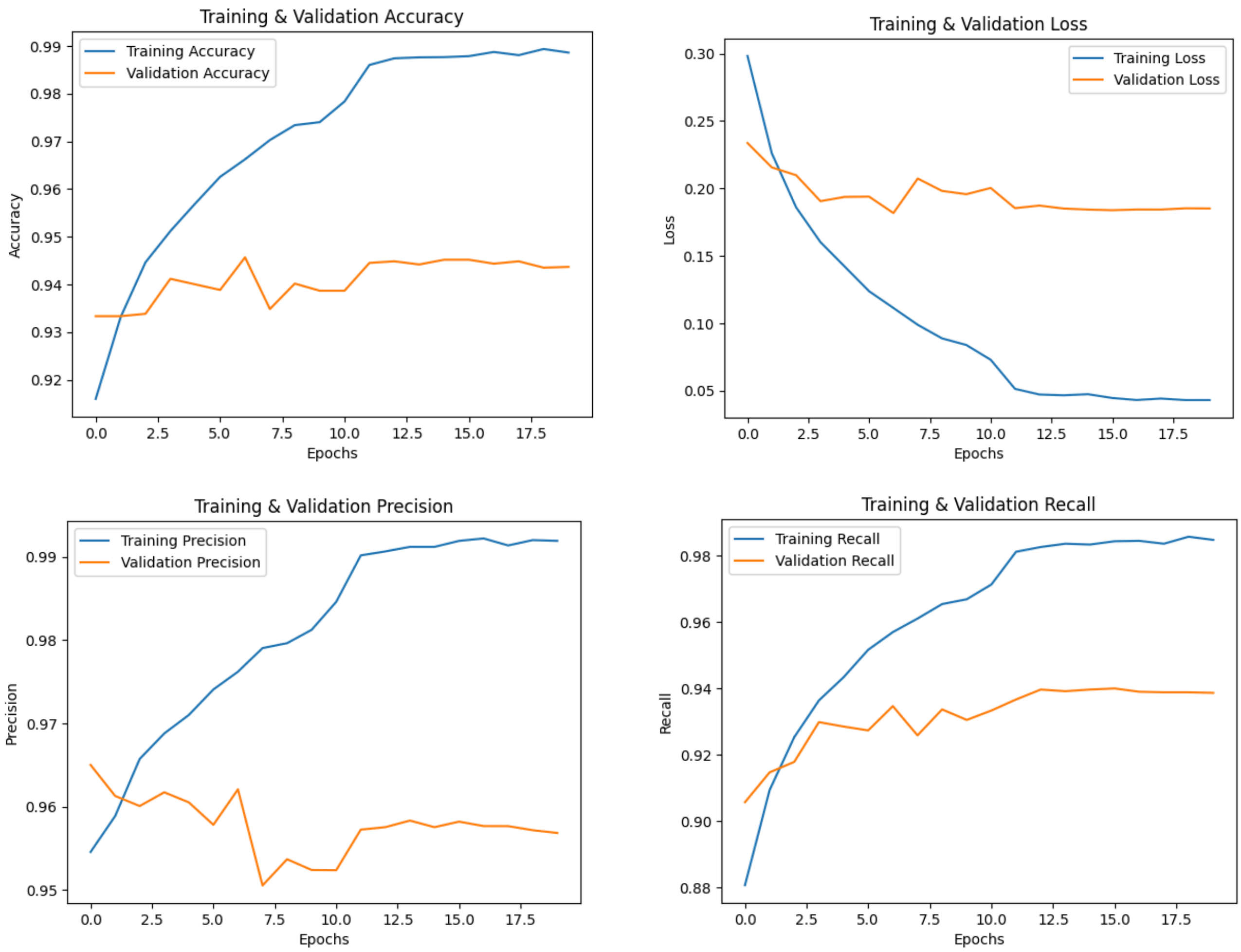
```
Total Training Time: 5095.24 seconds

Test Accuracy: 0.9661

Test Precision: 0.9754

Test Recall: 0.9605

              precision    recall  f1-score   support

           0       0.95      0.97      0.96       300
           1       0.92      0.93      0.93       300
           2       0.99      0.99      0.99       300
           3       0.99      0.98      0.98       300
           4       0.97      0.97      0.97       300
           5       0.94      0.94      0.94       300
           6       0.93      0.91      0.92       300
           7       0.96      0.95      0.95       300
           8       0.95      0.96      0.95       300
           9       0.98      0.95      0.96       300
          10       0.99      0.99      0.99       300
          11       0.93      0.95      0.94       300
          12       0.91      0.93      0.92       300
          13       0.98      0.98      0.98       300
          14       0.97      0.97      0.97       300
          15       0.96      0.95      0.96       300
          16       0.99      0.98      0.99       300
          17       0.98      0.97      0.97       300
          18       1.00      0.97      0.98       300
```

```
          19       0.98      0.98      0.98       300
          20       0.98      0.98      0.98       300
          21       0.97      0.99      0.98       300
          22       1.00      0.97      0.99       300
          23       0.96      0.98      0.97       300
          24       0.99      1.00      1.00       300

    accuracy                           0.97      7500
   macro avg       0.97      0.97      0.97      7500
weighted avg       0.97      0.97      0.97      7500
```

7. Visualization of Training Metrics

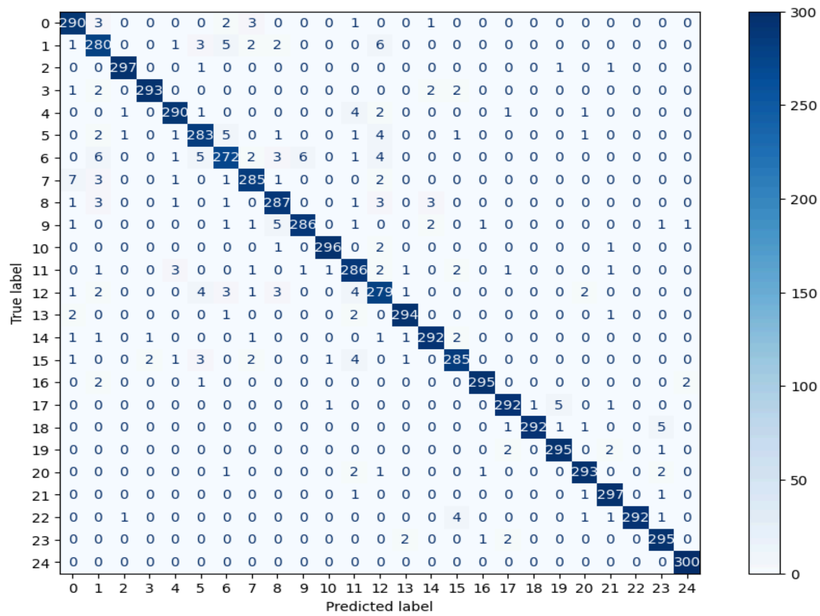
```
# Extract history
train_accuracy = history2.history['accuracy']
train_precision = history2.history['precision']
train_recall = history2.history['recall']
train_loss = history2.history['loss']

val_accuracy = history2.history['val_accuracy']
val_precision = history2.history['val_precision']
val_recall = history2.history['val_recall']
val_loss = history2.history['val_loss']

epochs = list(range(len(train_accuracy)))


# Plot Accuracy
plt.plot(epochs, train_accuracy, label='Training Accuracy')
plt.plot(epochs, val_accuracy, label='Validation Accuracy')
plt.title("Training & Validation Accuracy")
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()

# Plot Loss
plt.plot(epochs, train_loss, label='Training Loss')
plt.plot(epochs, val_loss, label='Validation Loss')
plt.title("Training & Validation Loss")
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()

# Plot Precision
plt.plot(epochs, train_precision, label='Training Precision')
plt.plot(epochs, val_precision, label='Validation Precision')
plt.title("Training & Validation Precision")
plt.xlabel('Epochs')
plt.ylabel('Precision')
plt.legend()
plt.show()

# Plot Recall
plt.plot(epochs, train_recall, label='Training Recall')
```

```
plt.plot(epochs, val_recall, label='Validation Recall')
plt.title("Training & Validation Recall")
plt.xlabel('Epochs')
plt.ylabel('Recall')
plt.legend()
plt.show()
```



8. Confusion Matrix

```
# Confusion Matrix
fig, ax =
plt.subplots(figsize=(12,8))
cm_matrix =
confusion_matrix(test_labels,
predictions)
cm_display = ConfusionMatrixDisplay(
            confusion_matrix=cm_matrix,
            display_labels=list
                (classes.values()))

cm_display.plot(cmap='Blues', ax=ax)
plt.show()
```



Conclusion

In this work, a comprehensive bird species detection system was designed and deployed, integrating real-time computer vision capabilities with intelligent AI agents for knowledge dissemination. The application demonstrates robustness across different media inputs — images, live captures, and videos — offering users an engaging and educational experience.

The project's modular structure allows future enhancements, such as support for additional bird species, improved model accuracy through fine-tuning, and integration with mobile applications. With the rising interest in citizen science and biodiversity studies, such an accessible and interactive platform holds the potential to significantly aid both enthusiasts and researchers in bird species identification and conservation awareness.